AFRL-RI-RS-TR-2009-166
Final Technical Report
June 2009

MULTICORE HARDWARE EXPERIMENTS IN SOFTWARE PRODUCIBILITY

University of Arizona

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-166 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
WILLIAM MCKEEVER
Work Unit Manager

/s/
EDWARD J. JONES, Deputy Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JUN 2009	Final	Dec 2007 – Dec 2008

4. TITLE AND SUBTITLE

MULTICORE HARDWARE EXPERIMENTS IN SOFTWARE PRODUCIBILITY

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-08-1-0024

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Jonathan Sprinkle and Brandon Eames

5d. PROJECT NUMBER
459T

5e. TASK NUMBER
AZ

5f. WORK UNIT NUMBER
MC

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Arizona
888 North Euclid Ave.
Tucson, AZ 85721-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RITB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2009-166

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2009-2918 Date Cleared: 29-JUN-2009

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This final technical report was prepared for the Air Force Research Laboratory, under award #FA8750-08-1-0024, titled "MultiCore Hardware Experiments in Software Producibility." This report details our findings when taking heterogeneous systems software, designed for a distributed environment, and running it on a single-core, and later multi-core, computers. Our research outcomes are significant, indicating that in our case, significant variance was seen in system performance, and that the variance increased with the number of cores used. We also created some strategies to reduce this variance, namely a weak time triggered infrastructure, which we imposed upon the system: this strategy significantly reduced the variance of behaviors when trading up to multicore processors.

15. SUBJECT TERMS
Multi-core, Real-time Systems, Testing, Software Modernization

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			William E. McKeever, Jr.
U	U	U	UU	28	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

List of Figures

List of Tables

Acknowledgements

This work analyzes software built with tremendous effort by the Sydney-Berkeley Driving Team, which entered the DARPA Urban Challenge in 2006. Team members who performed work tangential to that described in this effort include Ben Upcroft, Michael Moser, Alen Alempijevic, Ashod Donikian, Alexei Makarenko, Will Uther, Robert Fitch, Humberto Gonzalez, Esten Grøtli, Todd Templeton, Eric Chang, J. Mikael Eklund, Pannag R. Sanketi, David Johnson, Jan Biermeyer, Vason P. Srini, Christopher Brooks, Mark Godwin, and many others who donated their time and efforts.

Abstract

This final technical report was prepared for the Air Force Research Laboratory, under award #FA8750-08-1-0024, titled “MultiCore Hardware Experiments in Software Producibility.” This report details our findings when taking heterogeneous systems software, designed for a distributed environment, and running it on a single-core, and later multi-core, computers. Our research outcomes are significant, indicating that in our case, significant variance was seen in system performance, and that the variance increased with the number of cores used. We also created some strategies to reduce this variance, namely a weak time triggered infrastructure, which we imposed upon the system: this strategy significantly reduced the variance of behaviors when trading up to multicore processors.

1.0 Summary

We present the results of a series of experimental simulations of an autonomous ground vehicle, performed on various single core, and multicore processing platforms. Independent runs of the simulation (from the same initial conditions, on the same machine) produce discrete variances in the functional behavior, one of which is a “crash” where the simulated vehicle runs into a barrier, and cannot continue to function. Our simulations examine discrete differences in the success of avoiding this “crash”: namely, these machines with multiple cores are *more likely* to “crash” than machines with single cores. In fact, the more cores a machine has the *more likely* it is to “crash.” We provide some analysis of this phenomenon, including a hypothesis that the introduction of time-triggering for one particular component would significantly improve the chances of success. We conclude with the results of this lazy time-triggering, which verify our hypothesis, and some thoughts on how future work can reduce the functional variability in such simulations by design.

2.0 Introduction

Control of autonomous ground vehicles involves algorithms and design techniques from the disciplines of control, real-time systems, robotics, and software. As frequently observed in cyber-physical systems, the system designers may need experience in multiple areas, and knowledge of how control, communication, and computation interact in order to put together a system that behaves as expected.

In this effort, we approach the abstract scenario of a cyber-physical system that is "upgraded" from a single core processor to a multicore processor. Will the new system behave identically? Are certain observables likely to indicate fragility? In this approach, we examine the robustness of a somewhat fragile cooperation between two data-driven components capable of running on the same core, two different cores, or two different machines, when those components are deployed on various machines with various single or multicore processors.

Our domain focus, autonomous ground vehicles, grew out of a previous collaboration through the Sydney-Berkeley Driving Team. During that collaboration there was some empirical evidence of instability, and certainly vehicle behavior was sometimes erratic. Symptoms included chattering in the steering wheel, hesitant acceleration and braking, and unpredictable behaviors at certain areas of the course. Successive runs would occasionally have different behaviors, albeit only slight differences.

After the formal collaboration ended, we began to analyze the behavior of the ground vehicle in simulation, and fortuitously observed that a particular area of the course (we call it, informally, "dead-man's curve") occasionally resulted in the simulation ending abruptly, as the vehicle collided with a simulated barrier. After a few runs, we hypothesized that the same initial conditions could produce these two discrete behaviors on the same machine in *successive runs*. The discrete behaviors we see (crashing, and finishing) are exemplified in Figure 1.

What are the indicators? Without introducing the technical terms yet, some measurable discriminators between runs on the same machine produce some startling differences as the number of cores increase. That is, by comparing the planned paths (using naive comparison techniques) between two runs, we see that an increased number of cores results in less overall stability of the simulated vehicle. One example naive comparison (shown in Figure 2) compares the first planned waypoint at each timestep of the system's execution. However, differences in processor speed mean that faster machines will be able to execute some components (namely, those components that are dataflow triggered) more frequently, and will thus produce additional paths. This observation prompted us to ask the question: "How should we be comparing the behaviors of individual components of these cyber-physical systems, if not by their streamed output from the same initial conditions?"

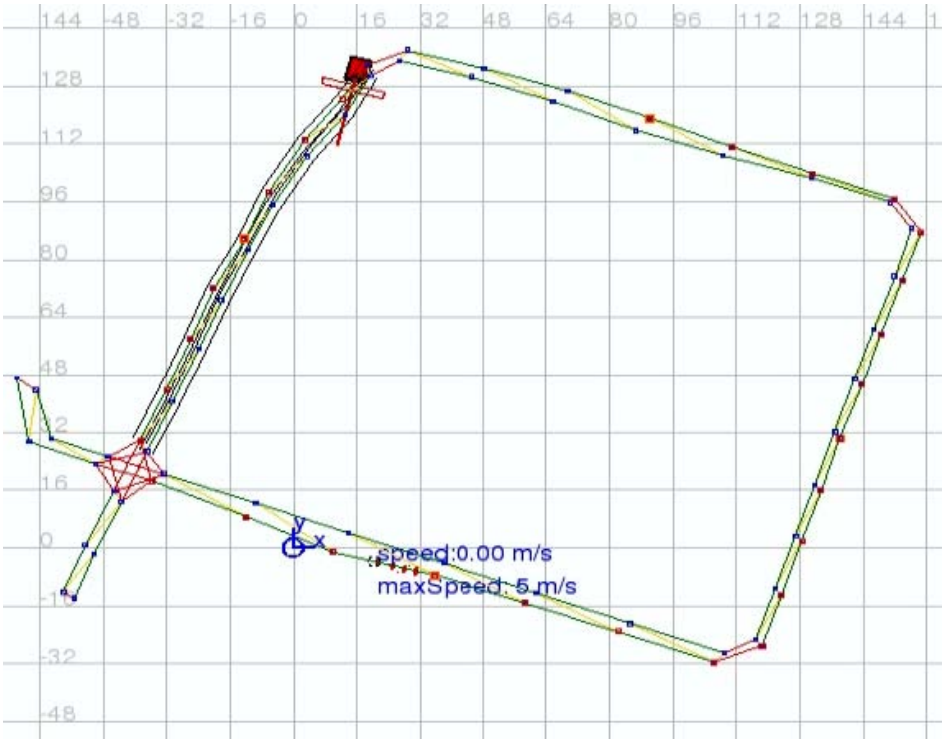
The answer to this question we believe is still open, and we do not answer it in this report. However, we do begin to analyze a particular discrete behavioral difference between various runs of an autonomous ground vehicle simulation: namely, that the number of times our vehicle fails to pass dead-man's curve should be a valid discriminator of the stability of our system across hardware changes. In this effort we ask:

- How can we improve the success rate?
- Are there integration strategies that mitigate the failure rate?
- Are there lightweight methods (that do not effect change in the binary code) to perform this reduction?

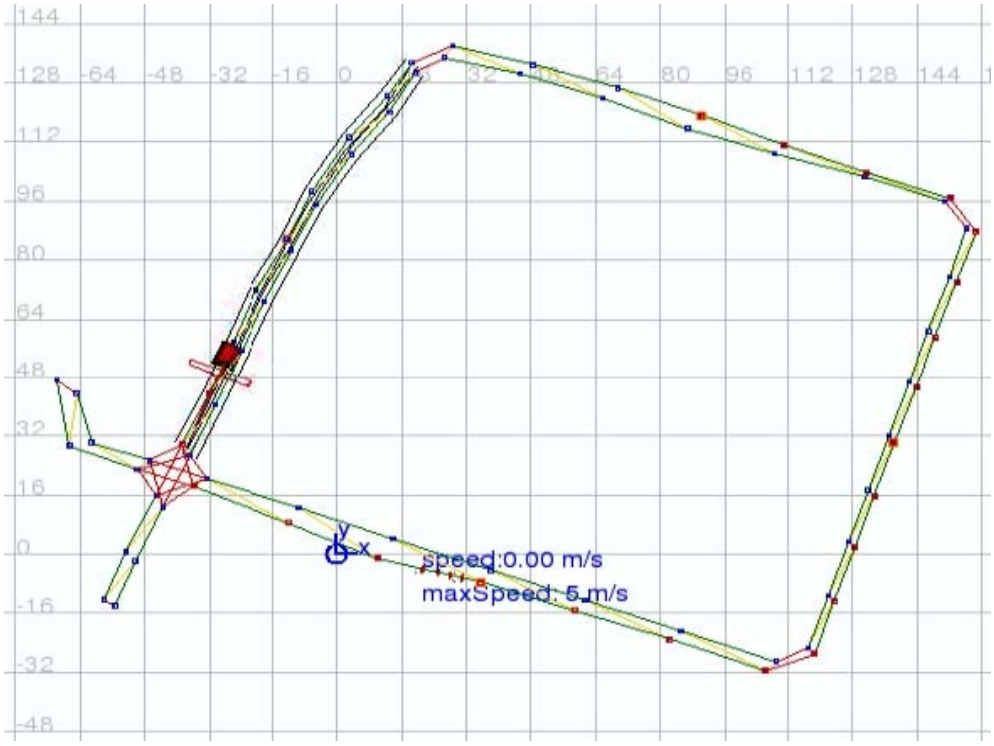
Each of these questions are addressed in this report and its analysis.

Finally, we address the question: What makes this system cyber-physical, and not just simulation? The difference is that these experiments, while fully simulated, present the challenges of the intersection of traditional computation, control, and communication as a joint problem. Computational tasks that are triggered by communication (i.e., dataflow) change when computational platforms change. This can affect the stability of the system from a control perspective. When the control stability is not certain, behavioral differences are amplified, which results in a change in computational (behavioral) output *from the same initial conditions*. In a rough sense, we have happened upon a chaotic system, where a change in computational platform increases the chaos.

Happily, we have also noticed that what was initially perceived as an intermittent software bug is actually a system integration flaw, now a logical error. This was only noticed because we amplified the problem sufficiently. We therefore posit that studying this example is more than just regressive analysis of one particular implementation, but is an applicable abstraction for future analysis. We further discuss this position in the future work section. Next, we discuss the control and computation tasks central to this system.



(a) The vehicle crashes into a simulated barrier.



(b) The vehicle passes the simulated barrier, and simulation terminates.

Figure 1: A crash, and a finish, of the simulation from the same initial conditions.

3.0 Methods, Assumptions, and Procedures

We are using the coupled algorithms of model-predictive control, and dynamic waypoint specification, as a basis with which to judge the fragility or robustness of our system to upgrade to multicore processors. Specifically, our legacy software, created using component-based design techniques, allows us to isolate these behaviors in single executables, which may operate multiple threads for their behavior.

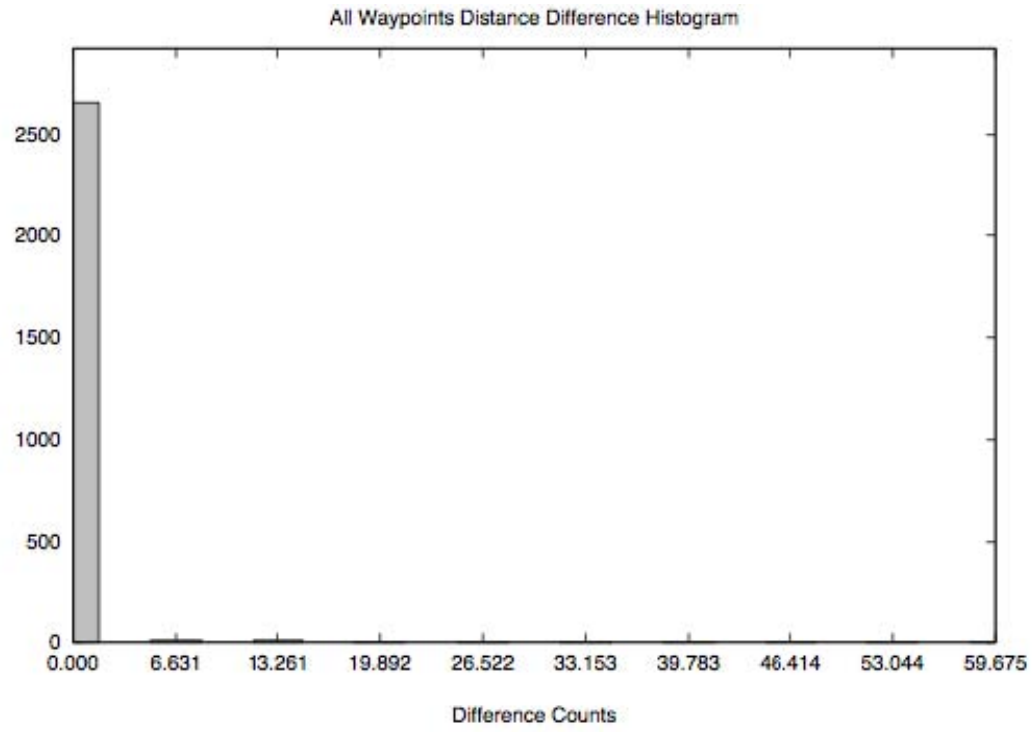
Generally, components for this system are developed by algorithm experts who understand well what their component should do computationally. Unfortunately, the behavior of composition of these components emerges, and is not engineered by design. A more principled system design might have considered the semantics of the composition of these heterogeneous components (as discussed in [1]), though such integration is difficult to enforce as a programming styleguide. It is much more appropriate to enforce this through interface calls, and "glue" code. We discuss this in the conclusion section as part of our future work.

Two important components under study are the local navigation component (dgclocalnav), and high level planner (highlevelplanner) , and each component is legacy source code written as part of the Sydney-Berkeley Driving Team, which was a joint entry into the DARPA Urban Challenge (i.e., autonomous ground vehicles) [2, 3]. Neither piece of software is modified from its use in that competition for any of these experiments.

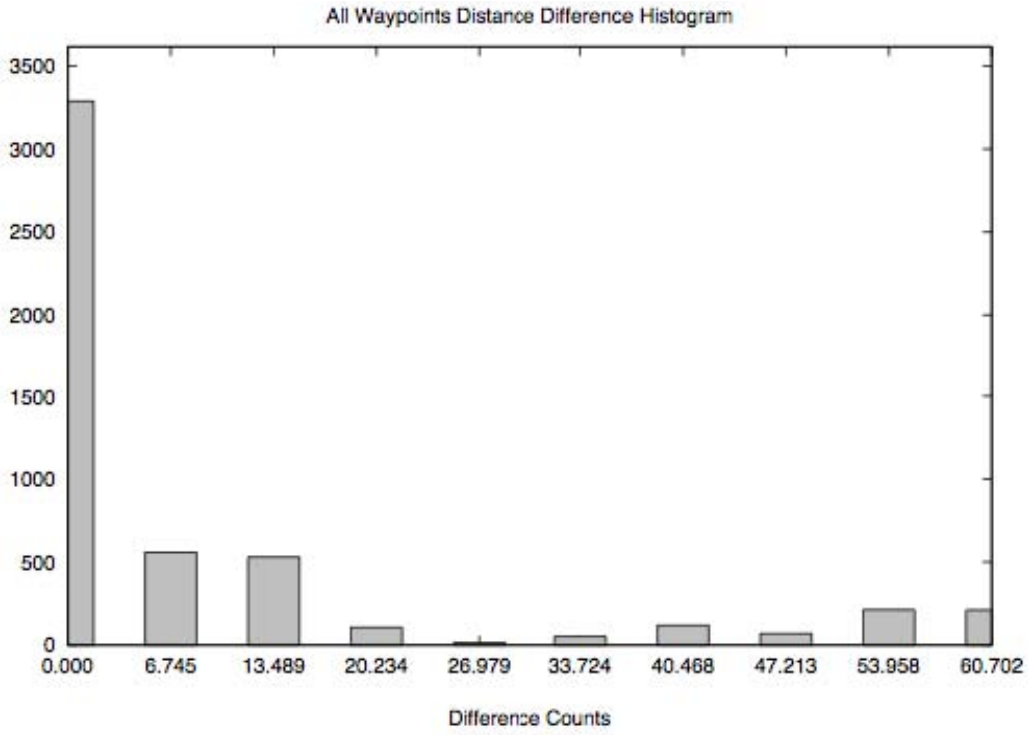
3.1 Local Navigation

The local navigation component (dgclocalnav) provides control inputs to the vehicle to follow a set of waypoints, while avoiding any obstacles not observed by street level navigation. The local navigation component receives a set of waypoints from highlevelplanner, and plans a path through various obstacles, and on the road, to follow that path. The component is extremely computationally intensive, and runs in multiple threads to gather input data, as well as process data as it is received. The internal behavior of this component utilizes model-predictive control [4] and previous work by the authors [5] has indicated suitability for use in real-time systems.

The inputs to the dgclocalnav component are the path plan (received from highlevelplanner, discussed next), the drivable grid (see [6]) that shows all local obstacles, the current state of the vehicle (*x, y, z, x', y',* , etc.), and other information such as the lanes and data from the vehicle odometer. Note that, perhaps not intuitively, the provided interface for the path plan actually receives information. This is contrary to the notion that a component provides data over a provided interface, and receives data over a required interface; this idiosyncrasy is by design.



(a) Two successive runs on a dual core machine, compared functionally.



(b) Two successive runs on a quad core machine, compared functionally.

Figure2: Variance between the functional behaviors of two successive runs seems to increase with the number of cores

The outputs from dgclocalnav are an array of vehicle inputs, which control the steering wheel angle, and the desired velocity of the vehicle, at future timesteps. The component can provide inputs over a predetermined horizon, say n seconds, and a sufficiently long horizon can affect the order of magnitude for which the highlevelplanner component (producing the set of goal waypoints) must execute in order to prevent the dgclocalnav from running out of desired waypoints.

Each waypoint received by dgclocalnav contains position and heading information, as well as tolerance levels for achieving the desired position. More formally, a waypoint w = ($x, y, \vartheta, d_{tol}, \vartheta_{tol}, v_{max}, t_{max}$), where ($x, y$) represents the desired position of the vehicle, θ represents the desired heading angle of the vehicle when it arrives at (x, y), d_{tol} and ϑ_{tol} represent tolerances on distance from (x, y) and heading, respectively, and v_{max} and t_{max} represent maximum allowable speed and turnrate, respectively, permitted to arrive at the desired position. When highlevelplanner is invoked, it determines, given the current actual position of the vehicle, a sequence of waypoints the vehicle should follow in order to proceed towards its goal and avoid obstacles.

Each such sequence is referred to as a plan. A plan $p = (w_0, w_1, ..., w_n)$ consists of up to 5 waypoints. highlevelplanner will emit a new plan each time it is invoked, regardless of whether the most recently emitted plan has been completely executed by the local navigation.

When simulating the vehicle's traversal around a rectangular course, the control software iteratively produces plans and attempts to carry out those plans. A run $R = (p_0, p_1, p_2, ..., p_m)$ is a sequence of plans produced by the high level planner during the simulation of a single traversal of the driving course. Depending on its execution rate and the rate at which the vehicle is moving, the highlevelplanner may emit successive plans which are nearly identical.

Alternatively, at times, successive plans can indicate vastly different positions on the course. During our analysis, in order to establish consistent results, we simulate the course traversal multiple times on a single machine, and collect the results for each run. A runset $RS = (r_0, r_1, r_2, ..., r_k)$ is a set of simulation runs, each of which executes from identical initial conditions over the same course.

Our approach seeks to isolate the impact of the computation platform on which the system executes on various observable results—namely, the runsets, and the final behavior of the system (crash, or success). We gather runsets from a variety of machines, from dual-core uni-processor machines to quad-core, dual-processor machines.

3.2 High-level Navigation and Planning

This component produces a set of up to 5 waypoints for dgclocalnav, with the goal of guiding the autonomous ground vehicle to certain checkpoints. We used this component in binary form (i.e., its sources are unavailable).

The highlevelplanner component subscribes to inputs of type gridmap (a local area that details obstacles), lanes (a series of boundaries that show where the road should begin and end), and considers the overall map of the drivable area, and the current objective waypoint (a so-called checkpoint, perhaps hundreds of meters away), in its calculation. Its output is a path plan, as discussed above, that avoids large obstacles, and solves the shortest path algorithm [7, 8] for the overall map to the next checkpoint. The path plan, as produced, generally is on the order of 1-10 meters, and directs the vehicle toward the next checkpoint by selecting a few nearby waypoints that will complete the path to the checkpoint.

For stability, running this component on the order of seconds, instead of milliseconds, will still provide waypoints for the vehicle, given that these waypoints are several meters ahead of the vehicle, which is traveling at a maximum of 3 m/s.

3.3 Experimental Setup

We present a brief discussion of the important simulation topics for this experiment. As the scenario under study is in the domain of autonomous ground vehicles, we have various robotics, vision, control, and navigation tasks communicating with one another. The legacy system under study has several framework choices which we inherit for study in this application.

3.3.1 Operating System and Middleware

All machines are running Kubuntu 6.10, and machines with multiple cores receive the multiprocessor scheduling, assignment, and preemption as defined in that kernel. We did not override any of those behaviors, and leave all decisions regarding such assignments and behaviors up to the kernel. To preserve some similarity across platforms, however, we ensure that the exact version of all machines (with respect to the software) is identical.

In order to facilitate a multi-computer (distributed) implementation for the physical system the system under simulation utilized the Ice middleware solution [9], version 3.2.0. All machines ran their own registry, and database exchange components.

3.3.2 MultiCore Hardware

For this analysis we utilized four classes of machines, as shown in Table 1. For clarity, we duplicate the OS information discussed previously in this table.

3.3.3 Waypoint Logging

The legacy system did not include the capability to log speculative plans (i.e., the planned waypoints which the local navigator needed to follow). We followed our policy of introducing new components, rather than modifying existing ones, and introduced a waypoint logger which logs all waypoint paths passed to the dgclocalnav component to a text file. The planning paths are ordered, but their time is not considered important. For each new path plan, the ordered set of waypoints is listed.

This logging permits regressive analysis, so that we can analyze the various idiosyncrasies of the system's behavior on different multicore processors. Recall that our goal is to reduce fragility of this system, and in order to reduce the fragility, we must have some metric against which we can compare our strategies.

Table 1: Machines utilized in this experiment

Machine Name	Processor Type	Processor Details	Operating System
macphee	Single Core	Allocated as one processor through VMWare. Hardware is Intel Core 2 Duo, 2.4 MHz	Linux Kubuntu 6.10 (Kernel 2.6.17-10-386), (VMWare running on Mac OS X Host)
geezer	Single Core	AMD Athlon 64 3000+, 1.98GHz 512KB Cache	Linux Kubuntu 6.10 (Kernel 2.6.17-10-386)
labmachine	Single Core	Intel Pentium 4 CPU, 3.00GHz 1MB Cache	Linux Kubuntu 6.10 (Kernel 2.6.17-10-386)
dimble	Dual Core	Intel Core2 Duo CPU, E4500, 2.20GHz 2MB L2 Cache	Linux Kubuntu 6.10 (Kernel 2.6.17-10-386)
beames-desktop	Quad Core	Intel Q6600, 2.4Ghz, 2x4MB L2 Cache	Linux Kubuntu 6.10 (Kernel 2.6.17-10-386)
feverstone	Dual Processor, Quad Core	Intel Xeon CPU, X5460, 3.16GHz, 2x6MB L2 Cache	Linux Kubuntu 6.10 (Kernel 2.6.17-10-386)
hardcastle	Dual Processor, Quad Core	Intel Xeon CPU L5420, 2.50GHz, 2x6MB L2 Cache	Linux Kubuntu 6.10 (Kernel 2.6.17-10-386)

3.3.4 Experiment Execution

We create a simulation instance, which runs for a period of time and terminates. During this time, the vehicle has opportunity to pass dead-man's curve and proceed down the home stretch. Our simulation time is 200 seconds from the time that all components have started up. A script creates components in the order of dependency (i.e., starting components in the correct order), and the script and all of its processes is forcefully killed after the appropriate time.

During the execution of this run, the important data which we analyze is recorded to a text file. The text file is named based on the machine name, and the times which that run were initiated.

A series of runs, varying between 50 and 500, depending upon machine strength and expected uptime, was performed on each machine, in order to mitigate the effect of spikes in the success rate. All components were executed on the same machine (i.e., a distributed simulation was not used).

3.4 Basic Simulation

The basic simulation is described in Figure 3. Components include the vehicle controller (car), localization component (faithlocaliser3d, imu), obstacle detection components (laser1, gridmap), and control and planning components (previously discussed).

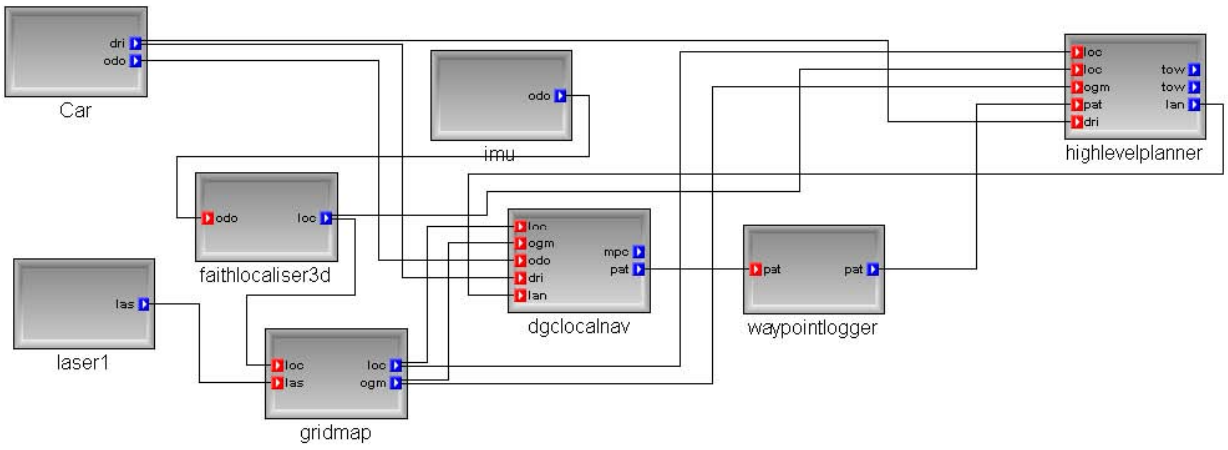


Figure 3: The basic simulation layout.

In this simulation, each component utilizes some heterogenous model of computation. For example, the laser1 component runs whenever it receives new laser input from the 3d simulation environment (Gazebo). Such updates come at the rate of the machine's free time, generally on the order of 20 Hz or more.

Because highlevelplanner performs blocking reads on its inputs, the receipt of inputs from the vehicle's state, the localization information, the drivability map, and the lanes information, the receipt of each of these items means that highlevelplanner will fire, and produce a new path. Each of the components feeding highlevelplanner data, however, runs at several Hz, which means that highlevelplanner will also run at several Hz. As we previously discussed, this frequency of execution is not necessary for stability.

In fact, this frequency of execution may introduce instability in the overall system. Our hypothesis is that reducing this frequency will measurably reduce the number of times that the system will crash. However, we first point to the number of successful runs by processor for this basic simulation.

In Figure 4 we see that the probability of success to pass dead-man's curve drops off dramatically as the number of cores increase. Detailed percentages are given in Table 2. Note that the two 8 core machines run at approximately 25% expected success, which is 1/3 of the success rate of the single core machine. Clearly, there is some impact on the number of cores on probability of passing dead-man's curve. We now introduce some methodology to reduce the frequency of highlevelplanner, with the expectation that this will improve the chance of success.

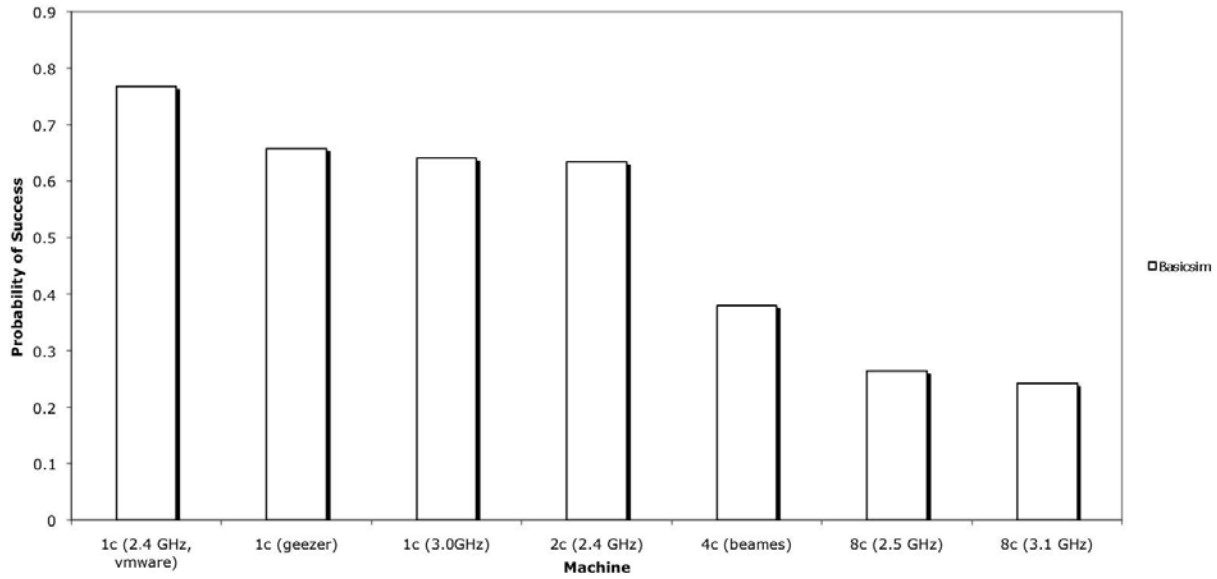


Figure 4: Rates of success for the basic simulation, across hardware platforms.

3.5 Logical Execution Time Simulations

The notion of logical execution time (LET) semantics is to provide a constant execution time for a process to execute in a real-time system, with constrained read and write behaviors. Languages such as Giotto [10] enforce LET semantics for all processes in the system. Solutions using LET semantics tend to scale well, as the logical release and read of outputs and inputs means (by design) that faster processors cannot introduce race conditions [11].

Table 2: Basic simulation success rate, by machine.

Machine Name	# Cores	Success Rate
macphee	1	0.7678
geezer	1	0.6567
labmachine	1	0.0.6400
dimble	2	0.6331
beames-desktop	4	0.3800
feverstone	8	0.2426
hardcastle	8	0.2645

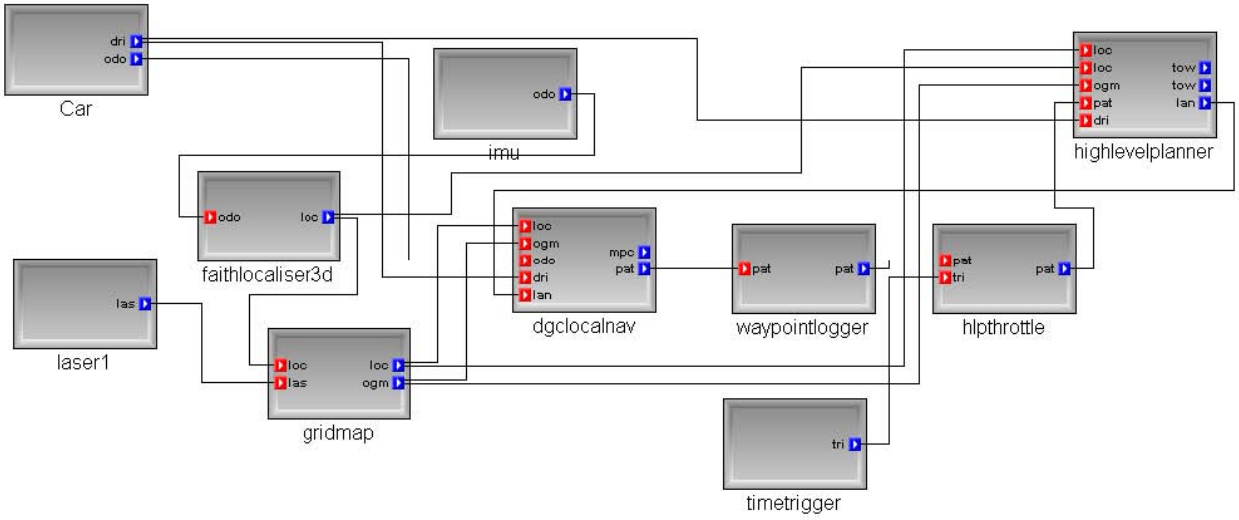


Figure 5: the letsim simulation layout.

In our system, the amount of code and the constrained operating system environment precludes a full LET semantics implementation. However, we hypothesize that a lazy form of LET semantics, where the frequency of execution of certain components is limited by time triggering (rather than data triggering through pure dataflow), will reduce the behavioral discrepancies (though it will not eliminate them).

We now introduce a new component, hlpthrottle, which is inserted between the waypointlogger and highlevelplanner. The semantics of hlpthrottle is that it fires every T seconds, and at that time, it performs a blocking read on its input interfaces, and fires once each of those interfaces is satisfied with new data. Once it fires, its output is produced in its provided interface. There are of course alternatives to these semantics, though for this application the frequency of execution of highlevelplanner is such that we can depend on a new waypoint path in much less than one second.

We are able to achieve a time-triggered behavior using the timetrigger component, whose fundamentals were out of the scope of this effort. In short, it uses POSIX messages to wake up at a specified time, or a set frequency, and upon wake-up it produces a message for all components subscribed to it. So, in fact, the hlpthrottle subscribes to timetrigger, and blocks on receipt of a trigger, and then blocks on receipt of a waypoint path. For these experiments, we use T = 5[s].

There are several advantages to obtaining time-triggered behavior by inserting a new component between our existing components, rather than modifying existing component code to perform a blocking read on the trigger. We maintain binary compatibility with each of our existing components, including any subtleties in their processing. Each of these components, therefore, will not modify its current methodology of receiving data. Ongoing processing will continue, and components that produce a refined view of the world based on new data will continue refining their data, even though snapshots of the refined world may change.

There are also disadvantages, in that we are increasing processor utilization (by adding a new component), and existing utilization by each component is maintained. However, we look past these subtleties for now, and examine the results of these simulations across several hardware platforms.

In Figure 6 we still see a drop off as the number of cores increases, but it is not as dramatic as with the basicsim simulation. In fact, we can compare the success rate of the letsim simulations to that of basic simulations, and see that we improve the probability of success dramatically. So, why do we see such a dramatic reduction in failures? In order to address this question, and justify our hypothesis, we provide some analysis of these results.

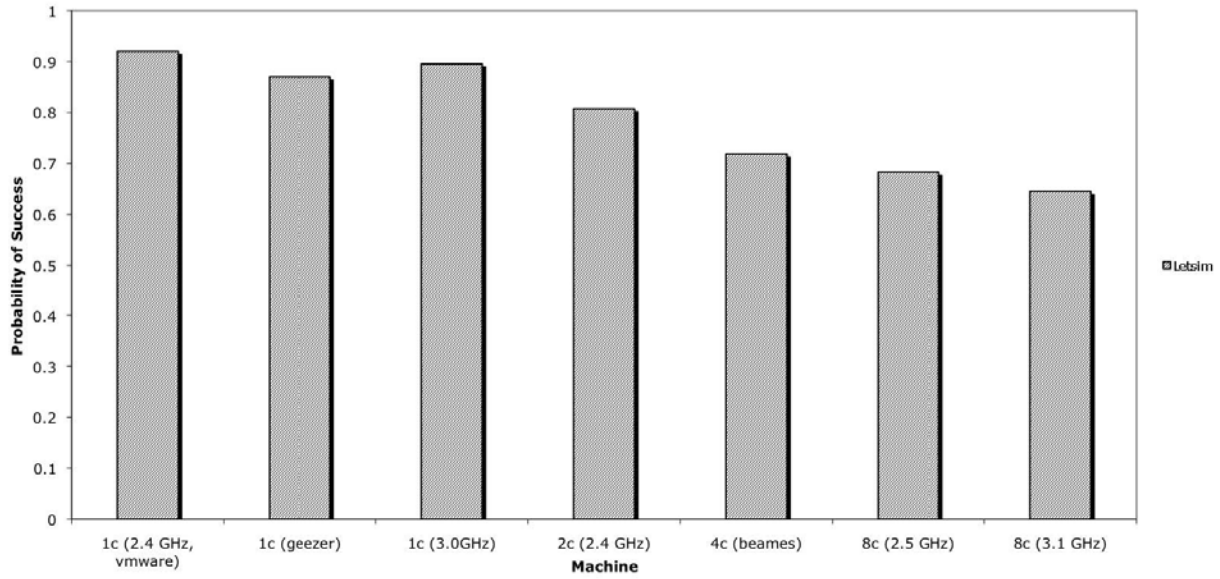


Figure 6: Rates of success for the letsim simulation, across hardware platforms.

Table 3: Logical Execution Time simulation success rate, by machine (compared to basic simulation success rate, by percent improved).

Machine Name	# Cores	Success Rate	Improvement %
macphee	1	0.9200	19.82
geezer	1	0.0.8700	32.48
labmachine	1	0.0.895	39.84
dimble	2	0.8074	27.53
beames-desktop	4	0.7185	89.07
feverstone	8	0.6450	165.86
hardcastle	8	0.6831	158.26

4.0 Analysis and Results

In order to better understand the behavior of the vehicle simulation, we plot and examine simulation results from a variety of perspectives. We have two primary simulation modes to consider: “basic” simulation, referred to as basicsim, and the simulation integrating the LET semantics, referred to as “letsim”. We have executed multiple runs of the simulations across several processors, as described previously. Figure 7 provides a plot of all waypoints generated over 56 runs in a basicsim simulation executed on the macphee machine. We partition each plan by waypoint and collect waypoint classes to form distinct plot series. A first-order analysis indicates that the waypoints fall within the same range, covering the path of the driving course. The initial position of the vehicle on the path is at approximately (20, 0), and proceeds in a counter-clockwise direction.

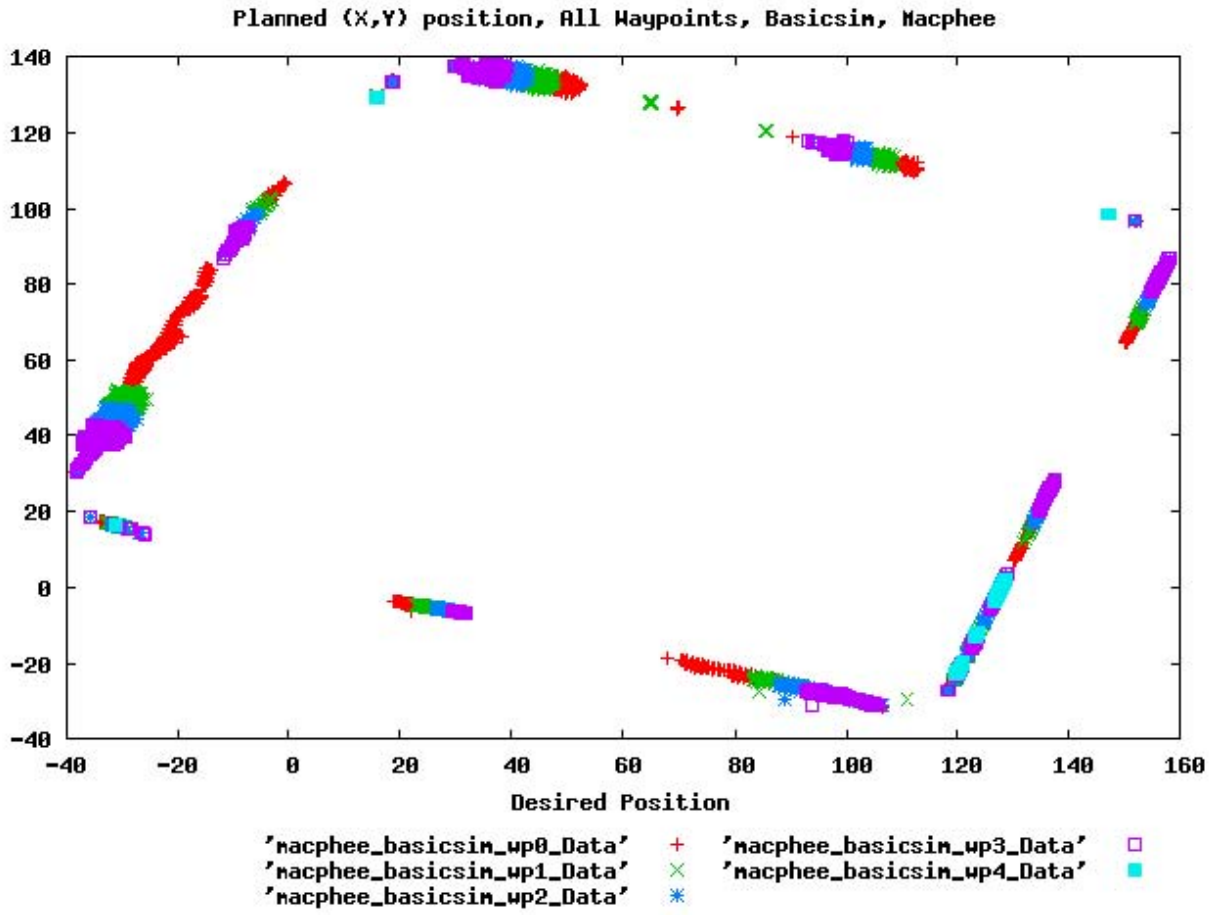


Figure 7: All plans (p =(w_0,...,w_N)) for every run of the basic simulation on machine macphee. Each w_i is shown in a unique representation.

Dead-man's curve is located at the upper-left corner of the course, ending at approximately position (20, 135) (refer to Figure 1a.). The plot shows distinct areas on the course path absent of waypoints in any plan on any simulation run. The highlevelplanner plans in reverse, using a checkpoint as a destination, and synthesizing intermediate waypoints to direct the vehicle to this location. Some maneuvers require dense planning, with waypoints being placed close together. Other maneuvers, once committed, require little planning, such as the completion of a turn or traversing a straight path. Also observable from the plot are, in some places, wide bands of waypoints, within and crossing series. This run-to-run variance in waypoint placement illustrates stability issues in the controller, which cascades as state changes, requiring varying path plans across runs. The impact of computational platform and power on stability indicates the cyber-physical reality of these issues.

Figure 8a plots the position of the first waypoint, w0, of each plan over all simulation runs executed on our five different target machines. The significant variance observed over the simulation course in Figure 7 is not as pronounced in this plot. This is due to the fact that this figure examines only w0, the initial waypoint in each plan. A careful examination of the plot also reveals that not all series result in waypoint placement along the same locations in the course. Different simulation runs on different machines produce similar, but not identical results. Further, although the inter-series variance is not as pronounced, variance can be observed, particularly in the course section leading to dead man's curve, and the path following dead-man's curve.

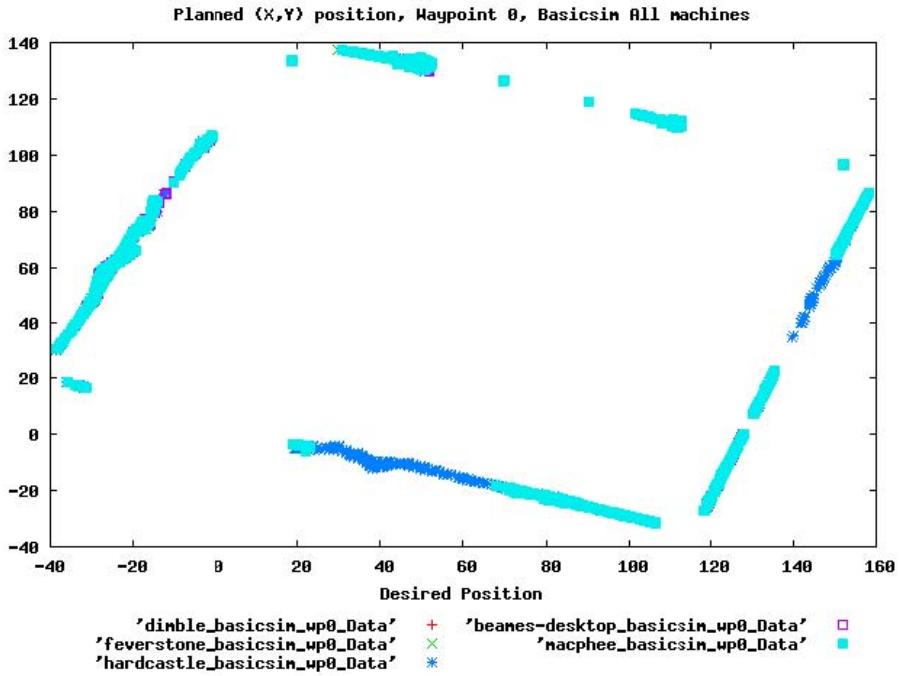
Figure 8b repeats the simulation from Figure 8a, but with the application of letsim semantics. One major impact of the application of letsim is a significant reduction in the number of waypoints processed during the simulation. The reduction is due to the fact that the hlpthrottle component only allows waypoints to be emitted to the dgclocalnav at appropriate times. Only those emitted waypoints are logged and processed. Immediately obvious from the plots is the lack of inter-series variance. We draw two distinct conclusions from the letsim plots. First, letsim simulation significantly reduces inter-run variance on a single machine. Second, letsim reduces inter-machine variance. The primary contention discussed above is that the execution platform running the simulation directly impacts the controller stability. The lack of variance in this plot indicates that the inclusion of letsim has significantly reduced the impact of machine platform on variance.

Figure 9 illustrates all waypoints from all runs of a letsim simulation executing on the beames-desktop platform. Similar results are indicated, with significantly reduced inter-run variance.

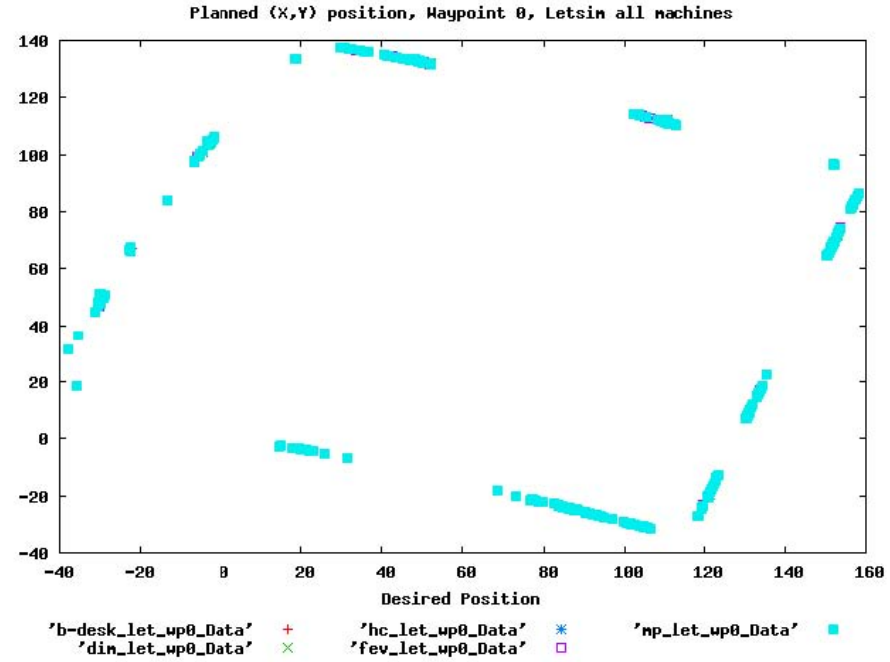
5.0 Conclusions

Admittedly, these experiments show that the presented design is somewhat fragile. A critical look at the use of time-triggering could argue that addressing the fragility through more sophisticated data-driven models is appropriate. We posit that such an approach may still lack robustness across multiple hardware platforms unanticipated in the future.

Our work in this effort does not address the redevelopment of algorithms, but rather their integration in such cyber-physical systems, where the physical platform's behavior will *always* have some behavioral variances. In our case, the fact that intrinsic behavioral differences introduce functional differences shows that our simulations show the same kinds of variability that we expect from the physical platform, and the fact that we are decreasing functional differences using this integration scheme should give higher confidence that such schemes are appropriate for the design of cyber-physical systems.



(a) All basic simulations.



(b) All letsim simulations.

Figure 8: Difference in w0 of all path sets across all machines, based on the basic simulation (no throttling) and the letsim (time triggered behavior).

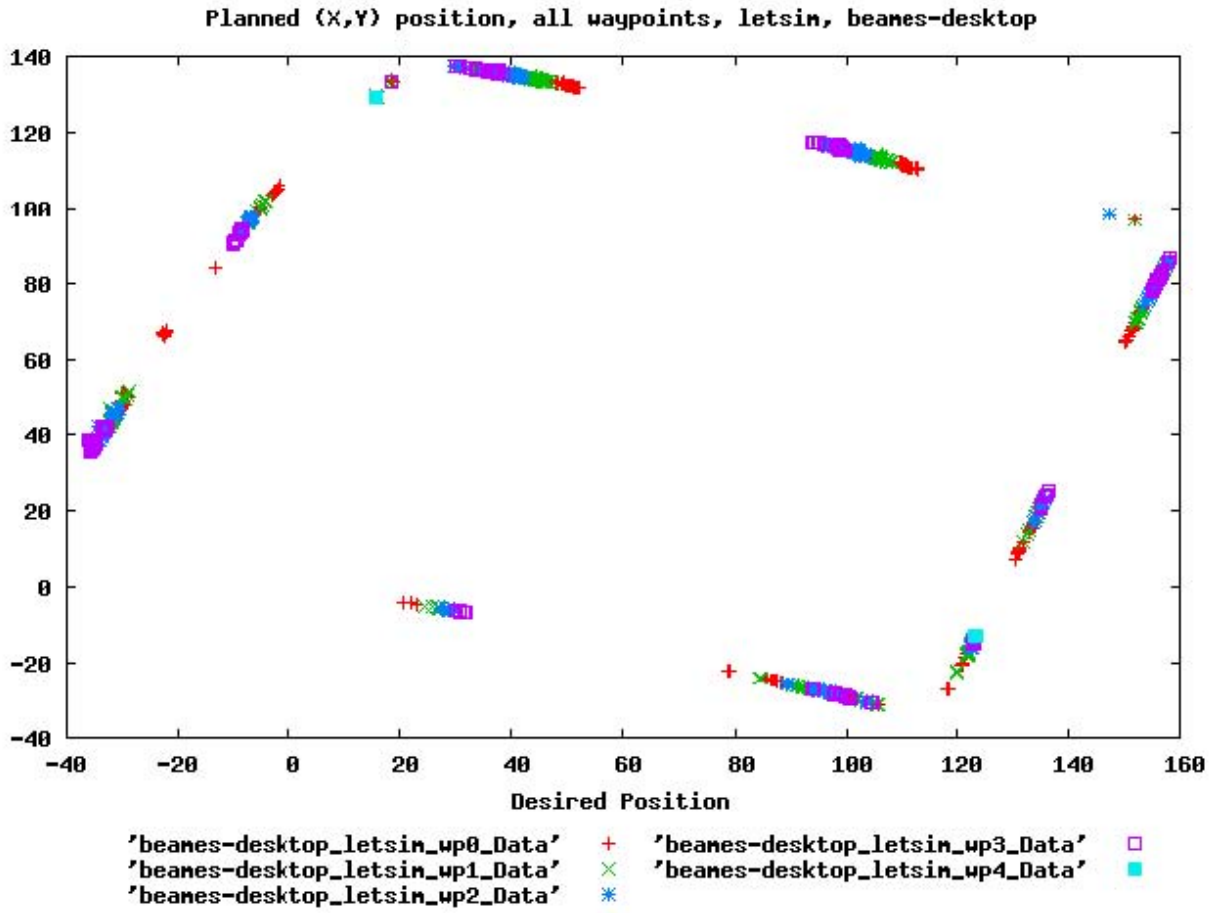


Figure 9: All plans ($p = (w_0,...,w_N)$) for every run of the letsim simulation on machine beames-desktop. Each w_i is shown in a unique representation. Compare to Figure 7.

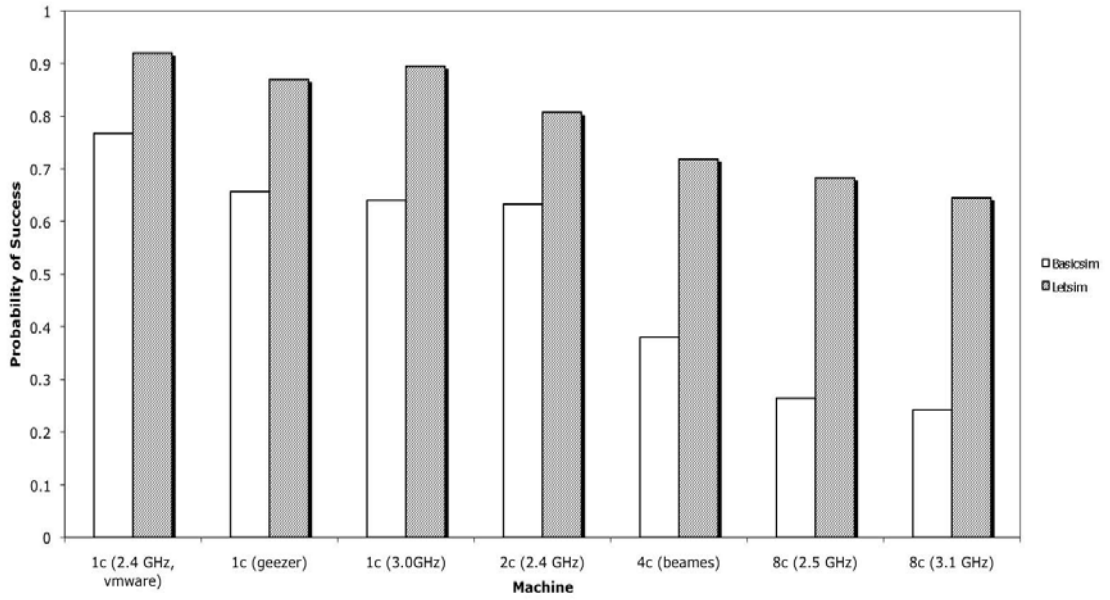


Figure 10: Success and failure rates by processor architecture

- Can we improve the success rate? Yes. Our success rate improved for all classes of hardware, and for classes of hardware with more core, we saw significant improvements (up to ≈150%).
- Are there integration strategies that mitigate the failure rate? Yes. We showed that a lazy form of time-triggering dramatically mitigated this failure rate.
- Are there lightweight methods (that do not effect change the binary code) to perform this reduction? Yes. For middleware-integrated systems, the introduction of new "throt¬tling" components can intercept, and pass along, data in a more controlled manner than the original designers intended.

We see a rich agenda of future research in utilizing this example in the abstract. How can two behaviors, measured on significantly different computational platforms, and using the same initial conditions, be compared for equivalence? What epsilon differences in behavior are sufficiently similar? Can chaotically timed dataflow components be tamed with lightweight time-triggered throttling?

There are also application-specific questions that can now be asked. Is the use of a 5¬second period for this component the optimal? What analysis can be done to determine the necessary frequency? Is this frequency tied to the vehicle platform under simulation and its fundamental architecture, and/or to the control strategy used to control that platform? Answering such questions can address semantic issues of how future code can be generated to appropriate values based on a target vehicle platform. Additional work in code generation can synthesize integration code that enforces a particular model of computation for the integrated heterogeneous system.

6.0 References

[1] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, and C. Goble. Heterogeneous composition of models of computation. Technical Report UCB/EECS-2007-139, EECS Department, University of California, Berkeley, Nov 2007. An earlier version of this paper was published in ICCS. A later version has been accepted for publication in Future Generation Computer Systems (FGCS), Elsevier.

[2] B. Upcroft, M. Moser, A. Makarenko, D. Johnson, A. Donikan, A. Alempijevic, R. Fitch, W. Uther, J. Biermeyer, H. Gonzalez, E. I. Grøtli, V. P. S. Todd Templeton, and J. Sprinkle. DARPA Urban Challenge Technical Paper: Sydney-Berkeley Driving Team. Technical report, University of Sydney; University of Technology, Sydney; University of California, Berkeley, June 2007.

[3] B. Upcroft, A. Makarenko, M. Moser, A. Alempijevic, A. Donikian, W. Uther, and R. Fitch. Empirical evaluation of an autonomous vehicle in an urban environment. Journal of Aerospace Computing, Information, and Communication, 4(12):1086–1107, Dec. 2007.

[4] F. Allgower and A. Zheng, editors. Nonlinear Model Predictive Control, volume 26 of Progress in Systems and Control Theory. Birkhauser Verlag, Basel-Boston-Berlin, 2000.

[5] J. Sprinkle, J. M. Eklund, H. J. Kim, and S. S. Sastry. Encoding aerial pursuit/evasion games with fixed wing aircraft into a nonlinear model predictive tracking controller. In Proceedings of the 43rd IEEE Conference on Decision and Control, volume 3, pages 2609–2614, December 2004.

[6] A. Elfes. Occupancy grids: A stochastic spatial representation for active robot perception. In Proc. 6th Conference on Uncertainty in AI, 1989.

[7] E. W. Dijkstra. A note on two problems in connexion with graphs. Numer. Math., 1:269–271, 1959.

[8] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2):100– 107, 1968.

[9] M. Henning. A new approach to object-oriented middleware. Internet Computing, IEEE, 8(1):66–75, Jan-Feb 2004.

[10] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: a time-triggered language for embedded programming. Proceedings of the IEEE, 91(1):84–99, Jan 2003.

[11] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for dis¬tributed giotto. SIGPLAN Not., 40(7):21–30, 2005.